Enrolment No./Seat No_______________

GUJARAT TECHNOLOGICAL UNIVERSITY

BE - SEMESTER–IV (NEW) EXAMINATION – SUMMER 2024

Subject Code:3140705 Date:08-07-2024

Subject Name: Object Oriented Programming -I

Time:10:30 AM TO 01:00 PM Total Marks:70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

MARKS

Q.1 **(a)** Answer in one word:(one mark for each) **03**

I. Which integral type in Java has the exact range from -2147483648 (-2^{31}) to 2147483647 ($2^{31}-1$).

II. The software for developing and running Java programs.

III. It is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM).

(b) Write a program that reads an integer and adds all the digits in the integer. For example, if an integer is 932, the sum of all its digits is 14. **04**

Hint: Use the % operator to extract digits, and use the / operator to remove the extracted digit. For instance, 932 % 10 = 2 and 932 / 10 = 93.

(c) Analyze following code pieces (assume they are properly enclosed in class and main function) and write the output/error (No justification is required): (one mark for each) **07**

```
I. int num = 5;
        int x = 1;
        for (int i = 1; i <= num; i++) {
            x *= i;
        }
  System.out.println("x is " + x);
```

```
II. int[] nums = {1, 5, 3, 9, 7};

        int m= 1;
        int s= 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] > m) {
                s = m;
                m = nums[i];
            } else if (nums[i] > s && nums[i] != m) {
                s = nums[i];
            }
        }
        System.out.println("s is: " + s);
```

```
III. int i=1,j=0,k=0;
            if (i > k) {
if (j > k)
System.out.println("i and j are greater than k");
}
else
System.out.println("i is less than or equal to k");
IV. int  x = 3, y = 3;
switch (x + 3) {
case 6: y = 1;
default: y += 1;
}
System.out.println("y is " + y);
V.  class HelloWorld {
            public static void main(String[] args) {
            int[] numbers = new int[1];
            numbers[0]=0;
            m(numbers);
            System.out.println("numbers[0]          is"          +
numbers[0]);
            }
            public static void m(int[] y) {
            y[0] = 3;
            }
}
VI. StringBuffer s1 = new StringBuffer("Complete");
        s1.setCharAt(1,'i');
        s1.setCharAt(7,'d');
        System.out.println(s1);
VII.  int Integer = 24;
char String  ='I';
System.out.print(Integer);
System.out.print(String);
```

Q.2 **(a)** Describe the relationship between an object and its defining class. **03**

(b) Write java statements for following class: **04**

```
public class Counter {
 int c;
}
```

i. Declare a parameterized constructor to initialize variable c.

ii. Declare instance method to increment c and return updated value of c.

(c) Justify following: **07**

i. In general, constructors should be public.

ii. Why main is declared as static.

iii. Why we do not declare destructors in java.

iv. There is no pointer in java.

OR

(c) Differentiate between following: **07**

i. this keyword vs super keyword

ii. abstract class vs interface

Q.3 **(a)** Give definition of method overloading. **03**

(b) i. Analyze following code pieces and write the output/error and briefly justify output: **04**

```
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  @Override
  public String getInfo() {
    return "Student";
  }
}

class Person {
  public String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

ii. Given the following source code, which comment line can be uncommented without introducing errors?

```
abstract class MyClass {
abstract void f();
final void g() {}
//final void h() {} // comment (1)
}
final class MyOtherClass extends MyClass {
public static void main(String[] args) {
MyClass mc = new MyOtherClass();
}
void f() {}
void h() {}
 //void g() {} // comment (2)
}
```

(c) Discuss public, private, protected and default access modifiers with examples. **07**

OR

Q.3 **(a)** What is the final keyword? Give different uses of the final keyword. **03**

(b) Declare an interface called Function that has a method named evaluate that takes an int parameter and returns an int value. **04**
Create a class called Half that implements the Function interface. The implementation of the method evaluate() should return the value obtained by dividing
the int argument by 2.

(c) What is an Exception? Explain Exception handling in JAVA with the help of example. **07**

Q.4	**(a)**	How do keyword throw differ from throws in Exception handling?	**03**
	(b)	Differentiate between a byte oriented and a character-oriented stream.	**04**
	(c)	Explain binary I/O classes. Demonstrate java file I/O with any I/O class to read a text file.	**07**

OR

Q.4	**(a)**	Compare JavaFX with Swing and AWT.	**03**
	(b)	i. List any four UI controls in JavaFX. ii. List any two Panes for Containing and Organizing Nodes	**04**
	(c)	Write a program to demonstrate GUI application development using JavaFX.	**07**
Q.5	**(a)**	What are the differences between ArrayList and LinkedList? Which list should you use to insert and delete elements at the beginning of a list?	**03**
	(b)	What is Java Collection Framework? List four collection classes?	**04**
	(c)	Give the name of class or interface that provides following facility (Do not elaborate)(one mark for each) i. Dynamic array to store the duplicate element of different data types. It maintains the insertion order. ii. Implements the last-in-first-out data structure. iii. Implements the first-in-first-out data structure. iv. Implements the first-in-first-out data structure and it holds the elements or objects which are to be processed by their priorities. v. It represents the unordered set of elements which doesn't allow us to store the duplicate items. vi. It is a red-black tree-based implementation. It provides an efficient means of storing key-value pairs in sorted order. vii. It allows us to store key and value pair, where keys should be unique.	**07**

OR

Q.5	**(a)**	Explain thread life cycle.	**03**
	(b)	What is the package? Explain steps to create a package with example.	**04**
	(c)	Declare a class called Employee having employee_Id and employee_Name as members. Extend class Employee to have a subclass called SalariedEmployee having designation and monthly_salary as members. Define following: - Required constructors - Methods to insert, update and display all details of employees. - Method main for creating an array for storing these details given as command line arguments and showing usage of above methods.	**07**

****

Object Oriented Programming –I

SUMMER 2024

Q.1 (a) Answer in one word:(one mark for each) I. Which integral type in Java has the exact range from 2147483648 (-231) to 2147483647 (231-1). II. The software for developing and running Java programs. III. It is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM).

I. **int**
II. **JDK**
III. **Bytecode**

(b) Write a program that reads an integer and adds all the digits in the integer. For example, if an integer is 932, the sum of all its digits is 14. Hint: Use the % operator to extract digits, and use the / operator to remove the extracted digit. For instance, 932 % 10 = 2 and 932 / 10 = 93.

```
import java.util.Scanner;

public class SumOfDigits {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);


        System.out.print("Enter an integer: ");

        int num = sc.nextInt();


        int sum = 0;


        // Loop to extract and add digits

        while (num != 0) {

            int digit = num % 10;   // Extract last digit

            sum += digit;          // Add digit to sum

            num = num / 10;        // Remove last digit

        }
        System.out.println("Sum of digits = " + sum);


        sc.close();
    }
}
```

Input:

```
932
```

Output:

```
Sum of digits = 14
```

(c) Analyze following code pieces (assume they are properly enclosed in class and main function) and write the output/error (No justification is required): (one mark for each)

```
I. int num = 5;
int x = 1;
for (int i = 1; i <= num; i++) {
x *= i;
}
System.out.println("x is " + x);
```

```
II. int[] nums = {1, 5, 3, 9, 7};
int m= 1;
int s= 0;
for (int i = 0; i < nums.length; i++) {
if (nums[i] > m) {
s = m;
m = nums[i];
} else if (nums[i] > s && nums[i] != m) {
```

```
s = nums[i];
}
}
System.out.println("s is: " + s);
III. int i=1,j=0,k=0;
if (i > k) {
if (j > k)
System.out.println("i and j are greater than k");
}
else
System.out.println("i is less than or equal to k");
IV. int  x = 3, y = 3;
switch (x + 3) {
case 6: y = 1;
default: y += 1;
}
System.out.println("y is " + y);
V.  class HelloWorld {
public static void main(String[] args) {
int[] numbers = new int[1];
numbers[0]=0;
m(numbers);
System.out.println("numbers[0]
is"
numbers[0]);
}
public static void m(int[] y) {
y[0] = 3;
}
```

```
}
VI. StringBuffer s1 = new StringBuffer("Complete");
s1.setCharAt(1,'i');
s1.setCharAt(7,'d');
System.out.println(s1);
VII.  int Integer = 24;
char String  ='I';
System.out.print(Integer);
System.out.print(String);
```

I.

```
x is 120
```

II.

```
s is: 7
```

III.

(No output)

IV.

```
y is 2
```

V.

```
numbers[0] is3
```

VI.

```
Cimpleted
```

VII.

```
24I
```

Q.2 (a) Describe the relationship between an object and its defining class.

Relationship Between an Object and Its Defining Class

In Java, a **class** is a fundamental building block that acts as a *blueprint*, *template*, or *prototype* for creating objects.

It defines the data members (variables) and the methods (functions) that describe the behavior and properties of its objects.

An **object**, on the other hand, is a *concrete instance* created from that class using the `new` keyword.

The object holds its own copy of the instance variables defined in the class and can access all non-static methods of the class. While the class exists only once in the program as a logical structure, multiple objects can be created from the same class, each having independent state but sharing the same behavior.

Thus, the class provides the **definition**, and the object provides the **real usage** of that definition. Without a class, no object can be created, and without objects, the class remains an abstract description only. This demonstrates a strong and essential relationship where **class describes**, and **object represents**.

(b) Write java statements for following class:

public class Counter {

int c;

}

i.

Declare a parameterized constructor to initialize

variable c.

ii. Declare instance method to increment c and return

updated value of c.

For the class `Counter`, we need to write:

1. A **parameterized constructor** to initialize the instance variable `c`.
2. An **instance method** that increments `c` and returns the updated value.

---

i. Parameterized Constructor

A parameterized constructor allows us to **pass an initial value** to the object at the time of creation.
It assigns the received value to the instance variable `c`.

```
public Counter(int c) {
    this.c = c;   // initializes the instance variable using constructor
argument
}
```

---

ii. Instance Method to Increment and Return Updated Value

This method increases the value of `c` by 1 and then returns the new value.
It helps maintain the counter and track the updated state.

```
public int increment() {
    c = c + 1;   // increases counter by one
    return c;    // returns updated value
}
```

Final Answer

```
public class Counter {
    int c;

    // i. Parameterized constructor
    public Counter(int c) {
        this.c = c;
    }

    // ii. Method to increment c and return updated value
    public int increment() {
        c++;
        return c;
    }
}
```

(c) Justify following:

i. In general, constructors should be public.

ii. Why main is declared as static.

iii. Why we do not declare destructors in java.

iv. There is no pointer in java.

i. In general, constructors should be public.

Constructors are responsible for initializing objects, assigning memory, and setting the initial state of instance variables.

In most cases, we want objects of a class to be created freely by other classes, modules, or packages. When a constructor is declared **public**, it becomes accessible from any part of the program, ensuring easy object creation and promoting reusability of the class.

If constructors were private or protected, object creation would become restricted, reducing flexibility. Java frameworks, libraries, and application code typically require open access to create instances. Therefore, keeping constructors public supports **encapsulation, modularity, reusability, and proper class usability**, which are key features of object-oriented programming.

---

ii. Why main() is declared as static.

The `main()` method is the **starting point** of every Java application and must execute independently of any object. Declaring it as **static** ensures that the JVM can invoke the method **directly using the class name** without needing to create an object first. At program startup, no objects exist in memory, so a non-static main would be impossible to call.

A static main method also ensures:

- **Efficient execution**: No need to allocate memory for an object before starting the program.
- **Consistency**: JVM always knows where to begin execution.
- **Class-level access**: Static methods belong to the class, making them ideal for startup routines.

Thus, the main method is static to enable the JVM to launch the program in a reliable and object-independent manner.

---

iii. Why we do not declare destructors in Java.

Languages like C++ require destructors to manually release memory and clean up resources. Java, however, uses **automatic memory management** through a built-in **Garbage Collector (GC)**. The GC automatically identifies unused objects and frees memory without programmer intervention.

Because of this automated process, Java eliminates the need for user-defined destructors. This approach offers major advantages such as:

- **No memory leaks caused by forgetting to free memory**
- **No dangling pointers or invalid memory access**
- **Simplified programming model**
- **Increased safety and security**

The closest alternative is the `finalize()` method, but even this is managed by the JVM and not guaranteed to run immediately. Therefore, Java does not allow or need destructors, as the GC handles object cleanup.

---

iv. There is no pointer in Java.

Java intentionally avoids pointers to ensure **safety, security, and simplicity**. In pointer-based languages, programmers can directly access memory addresses, which may cause risks such as:

- **Buffer overflows**
- **Dangling pointers**
- **Accidental modification of critical memory**
- **Complex and error-prone code**

Instead of pointers, Java uses **references**, which behave like safe, controlled pointers managed internally by the JVM. This design hides low-level memory details and prevents the programmer from manipulating memory addresses directly.

Key advantages of no-pointers design:

- **Strong security** (prevents accessing unauthorized memory)
- **Automatic memory management**
- **Cleaner and easier syntax**
- **Platform independence**, since memory handling is JVM-controlled
- **Robust and error-free programs**

Therefore, Java excludes pointers to maintain simplicity, reliability, and high security.

(c) Differentiate between following: i. this keyword vs super keyword ii. abstract class vs interface.

Point of Difference	**`this` Keyword**	**`super` Keyword`**
1. Meaning	Refers to the current class object	Refers to the parent (super) class object
2. Purpose	Access current class variables, methods, constructors	Access parent class variables, methods, constructors
3. Resolves Ambiguity	Between **current class instance variables** and parameters	Between **parent and child class members** having same name
4. Constructor Call	`this()` is used to call **current class constructor**	`super()` is used to call **parent class constructor**
5. Must Be First Statement?	`this()` must be first statement in constructor	`super()` must be first statement in constructor
6. Inheritance Requirement	Does **not** require inheritance	Works **only** with inheritance

Point of Difference	`this` Keyword	`super` Keyword`
7. Method Calling	Calls current class method (`this.show()`)	Calls parent class method (`super.show()`)
8. Variable Access	Access current class variable (`this.x`)	Access parent class variable (`super.x`)
9. Returning Current Object	Can return the current object (`return this;`)	Cannot return parent object
10. Passing as Argument	Can pass current object as argument	Cannot pass parent object
11. Static Context	Cannot be used in static areas	Cannot be used in static areas
12. Object Reference	Refers to **current object only**	Refers to **parent part of current object**
13. Usage Type	Constructor chaining within same class	Constructor chaining between parent and child class
14. Polymorphism	Not directly linked to overriding	Used to access overridden parent class method

Abstract vs Interface:

Point of Difference	Abstract Class	Interface
1. Definition	A class that may contain abstract and non-abstract methods.	A collection of abstract methods (Java 7) or abstract + default + static methods (Java 8+).
2. Purpose	To provide a **partial implementation**.	To provide **full abstraction** and define a contract for classes.
3. Methods Allowed	Can have abstract, non-abstract, static, final, and constructor methods.	Can have abstract, default, static, and private methods (Java 8+). No constructors.
4. Variables	Can have instance variables, non-final variables allowed.	Only **public static final** (constants) by default.
5. Inheritance	A class can extend **only one** abstract class.	A class can implement **multiple interfaces** (supports multiple inheritance).
6. Access Modifiers	Methods can have any access modifier (public, protected, private).	All methods are **public** by default.
7. Constructors	Can have constructors.	Cannot have constructors.
8. Implementation Nature	Supports both **abstraction + code reuse**.	Supports **only abstraction**, no code reuse before Java 8.
9. Use of Keywords	Uses **extends** keyword.	Uses **implements** keyword.
10. Speed	Faster than interface (because no dynamic dispatch for all methods).	Slightly slower, because interface methods use dynamic dispatch.

Point of Difference	Abstract Class	Interface
11. Multiple Inheritance Support	Not allowed (single inheritance only).	Allowed (a class can implement many interfaces).
12. When to Use?	When classes share **common behavior and state**.	When classes share **only method signatures**.
13. Example	`abstract class Animal { abstract void sound(); }`	`interface Animal { void sound(); }`

Q.3 (a) Give definition of method overloading.

Method Overloading is a feature in Java where **two or more methods in the same class have the same name but differ in their parameter list**.
This difference may be in the **number of parameters**, **type of parameters**, or **order of parameters**.
It represents **compile-time polymorphism**, because the compiler decides which method to call based on the arguments passed.
Method overloading improves **readability**, **reusability**, and makes the code more intuitive.

Key Features of Method Overloading

- **Same method name**, but **different parameter list** (number, type, or order).
- Occurs **within the same class**.
- Represents **compile-time polymorphism**.
- Helps improve **code readability** and **reusability**.
- Return type **may be same or different**, but cannot be used alone to overload.

```
class Display {

  void show(int a) {

    System.out.println("Integer: " + a);

  }


  void show(String s) {

    System.out.println("String: " + s);

  }

}


public class Test {

  public static void main(String[] args) {

    Display d = new Display();

    d.show(5);
```

```
        d.show("Hello");
    }
}
```

(b) i. Analyze following code pieces and write the output/error and briefly justify output:

```
public class Test {
    public static void main(String[] args) {
        new Person().printPerson();
        new Student().printPerson();
    }
}

class Student extends Person {
    @Override
    public String getInfo() {
        return "Student";
    }
}

class Person {
    public String getInfo() {
        return "Person";
    }
    public void printPerson() {
        System.out.println(getInfo());
    }
}
```

ii. Given the following source code, which comment line can be uncommented without introducing errors?

```
abstract class MyClass {
    abstract void f();
    final void g() {}
    //final void h() {} // comment (1)
}


final class MyOtherClass extends MyClass {
    public static void main(String[] args) {
        MyClass mc = new MyOtherClass();
    }
    void f() {}
    void h() {}
    //void g() {} // comment (2)
}
```

i. Output and Justification

Output:

```
Person
Student
```

Justification:

- `new Person().printPerson();` calls `printPerson()` of `Person`. Inside `printPerson()`, `getInfo()` is called. Since the object is of `Person`, it executes `Person`'s `getInfo()`, printing **"Person"**.
- `new Student().printPerson();` creates a `Student` object. `printPerson()` is inherited from `Person`, but `getInfo()` is **overridden** in `Student`. At runtime, the JVM calls the overridden method due to **dynamic method dispatch**, so it prints **"Student"**.

ii. Valid Comment Line

- **Comment (1):** Cannot uncomment `final void h() {}` in abstract class because subclasses cannot override final methods.
- **Comment (2):** Cannot uncomment `void g() {}` because `g()` is **final** in `MyClass` and cannot be overridden.

✅ **Valid uncommentable line:** `void h() {}` in `MyOtherClass` — already provided in subclass; safe.

(c) Discuss public, private, protected and default access modifiers with examples.

---

1. public

Definition :
Members declared as `public` are accessible from **any class** in the program, whether in the **same package** or a **different package**. Public members have **no access restrictions**, making them ideal for methods and variables that need to be **globally accessible**. They are commonly used for **API methods, utility classes, or shared data**.

Key Points:

- Provides **unrestricted access**.
- Supports **cross-package usage**.
- Used for **methods that are intended to be reused**.

Code Example:

```
public class Example {
    public int data;
    public void display() {
        System.out.println("Data: " + data);
    }
}
```

---

2. private

Definition :
Members declared as `private` are **accessible only within the class** where they are declared. They cannot be accessed directly from other classes, including subclasses. This helps in **data hiding**, ensuring that internal class details are protected. Private members are essential for **encapsulation**, allowing controlled access through **public getter and setter methods**.

Key Points:

- Supports **encapsulation**.

- Restricts access to the class only.
- Prevents accidental modification from outside.

Code Example:

```
class Example {
    private int data;
    public void setData(int d) {
        data = d; // controlled access
    }
    public int getData() {
        return data;
    }
}
```

---

3. protected

Definition:
Members declared as `protected` are **accessible within the same package** and in **subclasses** even if they are in a different package. Protected access is useful for **inheritance**, allowing child classes to use or override parent class members while keeping them hidden from unrelated classes.

Key Points:

- Supports **inheritance**.
- Accessible **within package and by subclasses**.
- Provides **controlled exposure** for child classes.

Code Example:

```
class Parent {
    protected int data;
}

class Child extends Parent {
    void display() {
        data = 50; // accessible in subclass
        System.out.println(data);
    }
}
```

---

4. default (no modifier)

Definition:
If no access modifier is specified, it is considered **default** (also called **package-private**). Default members are **accessible only within the same package**. This is useful when you want to **share data or methods among classes in the same package** but prevent access from classes in other packages.

Key Points:

- Accessible **only within the package**.
- Not accessible from other packages.
- Supports **package-level encapsulation**.

Code Example:

```
class Example {
    int data; // default access
    void display() {
        System.out.println("Data: " + data);
    }
}
```

OR

Q.3 (a) What is the final keyword? Give different uses of the final keyword.

The `final` keyword in Java is a **modifier** used to impose restrictions on variables, methods, and classes.

When applied, it ensures that the entity **cannot be changed, overridden, or inherited**, depending on the context.

`final` promotes **immutability, security, and consistency** in the program. It is widely used in object-oriented programming to define **constants, immutable methods, and classes**, preventing accidental modification or misuse of critical data and functionality. The keyword helps maintain **robustness, reliability, and clarity** in code design.

Uses of `final` Keyword:

1. **Final Variable:**
   Declares a constant whose value cannot be changed after initialization.
   Ensures important data remains immutable throughout the program.
2. **Final Method:**
   Prevents a method from being overridden by any subclass.
   Helps maintain consistent behavior of critical methods.
3. **Final Class:**
   Prevents a class from being inherited by any other class.
   Useful for creating immutable or secure classes.
4. **Final Parameter:**
   Prevents a method parameter from being modified inside the method.
   Ensures the argument value remains constant during execution.
5. **Final Local Variable:**
   Declares a local variable that cannot be reassigned once initialized.
   Useful in anonymous classes or lambda expressions to maintain stability.

   Example:

   ```
   final class Demo {              // final class
   ```

```
    final int x = 10;              // final variable


    final void show(final int y) { // final method and final parameter
        // y = 5;  // not allowed
        System.out.println(x + y);
    }


    void test() {
        final int z = 20;          // final local variable
        System.out.println(z);
    }
}
 class Main {
    public static void main(String[] args) {
        Demo d = new Demo();
        d.show(5);
        d.test();
    }
}
```

(b) Declare an interface called Function that has a method named evaluate that takes an int parameter and returns an int value.

Create a class called Half that implements the Function interface. The implementation of the method evaluate()

should return the value obtained by dividing

the int argument by 2

Interface Declaration

```
interface Function {
    int evaluate(int x);
}
```

Class Implementing the Interface

```
class Half implements Function {

    @Override
    public int evaluate(int x) {
        return x / 2;
    }
}
```

(c) What is an Exception? Explain Exception handling in JAVA with the help of example.

What is an Exception?

An **exception** in Java is an **abnormal condition or unexpected event** that occurs during the execution of a program and **interrupts the normal flow of instructions**. Exceptions usually occur at **runtime** due to errors such as invalid input, division by zero, accessing an array beyond its limit, or failure to open a file. In Java, exceptions are treated as **objects** that represent error conditions. These objects are created and thrown when an error occurs, allowing the program to handle the error in a systematic manner

Need for Exception Handling

Without exception handling, a program terminates abruptly when an error occurs, which may cause **data loss** or **unexpected behavior**. Exception handling allows the program to **continue execution**, **display meaningful error messages**, and **take corrective actions**. It improves the **reliability, robustness, and user-friendliness** of the application.

Exception Handling in Java:

Exception handling in Java is a **powerful and structured mechanism** that enables programmers to **detect, handle, and manage runtime errors** efficiently. During program execution, various unexpected situations such as invalid input, arithmetic errors, or file access problems may occur. Java provides a **well-defined exception handling model** that allows such errors to be handled gracefully without terminating the program abruptly. This model improves **program reliability,**

readability, and maintainability. Java uses **five important keywords** to implement exception handling:

1. try

The `try` block is used to **enclose the code that may generate an exception**. It contains statements that are risky and could cause runtime errors. Whenever an exception occurs inside the `try` block, the normal execution stops and control is transferred to the appropriate `catch` block. A `try` block must be followed by either a `catch` block or a `finally` block.

---

2. catch

The `catch` block is used to **handle the exception** that occurs in the `try` block. Each `catch` block specifies the type of exception it can handle. When an exception occurs, the JVM searches for a matching `catch` block. If found, the exception is handled and the program continues execution. Multiple `catch` blocks can be used to handle different types of exceptions.

---

3. finally

The `finally` block contains code that **executes regardless of whether an exception occurs or not**. It is mainly used for **cleanup operations** such as closing files, releasing database connections, or freeing resources. The `finally` block ensures that important statements are always executed, making the program more secure and stable.

---

4. throw

The `throw` keyword is used to **explicitly throw an exception** from a method or block of code. It is generally used when the programmer wants to manually generate an exception due to a specific condition. By using `throw`, custom or predefined exceptions can be passed to the exception handling mechanism.

---

5. throws

The `throws` keyword is used in a **method declaration** to specify that the method may pass one or more exceptions to the calling method. It informs the caller that the method may cause exceptions that must be handled externally. This helps in **separating error-handling responsibilities** and makes the code cleaner and more organized.

Working of Exception Handling

1. Risky code is placed inside a `try` block
2. If an exception occurs, the JVM creates an exception object
3. The JVM searches for a matching `catch` block
4. The exception is handled and program continues
5. The `finally` block executes regardless of exception

Example:

```
public class ExceptionDemo {

   public static void main(String[] args) {

       int a = 20;

       int b = 0;

       try {

           int result = a / b;   // causes ArithmeticException

           System.out.println("Result: " + result);

       }

       catch (ArithmeticException e) {

           System.out.println("Error: Division by zero is not allowed");

       }

       finally {

           System.out.println("Execution completed");

       }

   }

}
```

OUTPUT:

```
Error: Division by zero is not allowed

Execution completed
```

Types of Exceptions

1. Checked Exceptions

- Checked at **compile time**
- Must be handled using try-catch or throws
- Examples: `IOException`, `SQLException`

2. Unchecked Exceptions

- Occur at **runtime**
- Not checked by compiler
- Examples: `ArithmeticException`, `ArrayIndexOutOfBoundsException`

Advantages of Exception Handling :

1. Prevents abnormal termination of the program.
2. Allows graceful handling of runtime errors.
3. Improves program reliability and stability.
4. Separates error-handling code from normal logic.
5. Makes programs easier to debug and maintain.

Q.4 (a) How do keyword throw differ from throws in Exception handling?

Basis of Comparison	**`throw`**	**`throws`**
Meaning	`throw` is used to **explicitly generate an exception** in a program.	`throws` is used to **declare exceptions** that a method may pass to the calling method.
Purpose	It is used when the programmer wants to **manually raise an exception** based on a condition.	It is used to **inform the caller** that a method may generate exceptions.
Usage Location	Used **inside the body of a method or block**.	Used in the **method declaration** (signature).
Type of Action	Actually **throws an exception object**.	Only **declares** the exception, does not throw it.
Number of Exceptions	Can throw **only one exception at a time**.	Can declare **multiple exceptions** separated by commas.
Responsibility of Handling	The exception must be **handled immediately** using try-catch or passed further.	The responsibility of handling the exception is **shifted to the calling method**.
Common Use	Used for **custom exceptions** or condition-based errors.	Used mainly with **checked exceptions**.
Effect on Program Flow	Immediately interrupts normal program execution.	Does not interrupt execution; only provides information.

(b) Differentiate between a byte oriented and a character oriented stream.

Basis of Comparison	**Byte-Oriented Stream**	**Character-Oriented Stream**
Definition	A byte-oriented stream is used to read and write data in the form of raw bytes, making it suitable for handling binary data.	A character-oriented stream is used to read and write data in the form of characters, which is ideal for text-based data.
Type of Data	It handles binary data such as images, audio files, video files, and executable files.	It handles textual data such as characters, strings, and text files.
Data Unit	The basic unit of data processed is a single byte of 8 bits.	The basic unit of data processed is a character, usually represented using 16-bit Unicode.
Encoding Support	Byte-oriented streams do not perform any character encoding or decoding during data processing.	Character-oriented streams automatically handle character encoding and decoding, making them suitable for international text.
Parent Classes	These streams are derived from the `InputStream` and `OutputStream` abstract classes.	These streams are derived from the `Reader` and `Writer` abstract classes.
Example Classes	Common examples include `FileInputStream`, `FileOutputStream`, `BufferedInputStream`, and `BufferedOutputStream`.	Common examples include `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.
Usage	Byte-oriented streams are mainly used when working with non-text files where data must be read exactly as stored.	Character-oriented streams are mainly used when working with text files where character representation matters.
Performance	These streams are generally faster when handling large amounts of binary data.	These streams provide better readability and convenience when handling text data.
Conversion Requirement	Byte streams require manual conversion if character data is needed.	Character streams internally convert bytes into characters automatically.
Error Handling	Errors related to binary data must be handled at the byte level.	Errors related to text data are handled at the character level, which is easier for programmers.

(c) Explain binary I/O classes. Demonstrate java file I/O with any I/O class to read a text file.

Binary I/O Classes in Java

Binary I/O classes in Java are used to **read and write data in the form of raw bytes**. These classes are useful when working with **binary files**, where data should not be altered or interpreted as characters. Examples of such files include **images, audio files, video files, PDF files, ZIP files, and executable files**.

Binary I/O works at the **byte level**, meaning it processes data as sequences of 8-bit bytes. This makes binary streams faster and more accurate for non-text data, because no character conversion or encoding is applied.

Binary I/O classes are provided in the **`java.io` package** and are based on two main abstract classes:

- **InputStream** – used to read binary data from a source
- **OutputStream** – used to write binary data to a destination

All binary input and output stream classes inherit from these two base classes.

Why Binary I/O is Needed

Binary I/O is required when:

- Exact data format must be preserved
- Data should not be modified during reading or writing
- Working with multimedia or system files
- Handling large files efficiently

Binary streams are faster compared to character streams because they do not perform character encoding or decoding.

Important Binary I/O Classes

1. FileInputStream

- Used to read bytes from a file.
- Reads one byte at a time or a block of bytes.
- Commonly used for binary file reading.

2. FileOutputStream

- Used to write bytes to a file.
- Can create a new file or overwrite an existing file.

3. BufferedInputStream

- Wraps another input stream.
- Uses a buffer to improve reading performance.

4. BufferedOutputStream

- Uses a buffer to store data before writing it to the file.

5. DataInputStream

- Allows reading primitive data types like int, float, double in binary form.

6. DataOutputStream

- Allows writing primitive data types in binary form.

Characteristics of Binary I/O Classes

- Operate on **bytes**
- Do not handle character encoding
- Faster for large data processing
- Suitable for non-text files
- More control over data storage

Java File I/O Using Binary Stream

Java provides file handling using binary I/O through **FileInputStream** and **FileOutputStream** classes. These classes allow reading and writing data directly to files in byte format.

Simple Java Program to Read a Text File Using Binary I/O

Even though text files are usually handled using character streams, binary streams can still read them as bytes.

```
import java.io.FileInputStream;
import java.io.IOException;

public class BinaryReadDemo {
    public static void main(String[] args) {

        try {
            FileInputStream fis = new FileInputStream("data.txt");
            int data;

            while ((data = fis.read()) != -1) {
                System.out.print((char) data);
            }

            fis.close();
        }
        catch (IOException e) {
            System.out.println("Error while reading file");
            }
        }
}
```

Advantages of Binary I/O Classes

- Faster file processing
- Efficient memory usage
- Suitable for large and binary files
- Precise control over file data

Disadvantages of Binary I/O Classes

- Not suitable for direct text manipulation
- Manual conversion needed for character data
- Less readable than character streams

OR

Q.4 (a) Compare JavaFX with Swing and AWT.

Basis	AWT (Abstract Window Toolkit)	Swing	JavaFX
Introduction	AWT is the oldest Java GUI toolkit and was introduced in the early versions of Java.	Swing was introduced later to overcome the limitations of AWT.	JavaFX is the latest GUI framework introduced to build modern Java applications.
Component Type	Uses **heavyweight components** that depend on the native operating system.	Uses **lightweight components** that are mostly independent of the OS.	Uses **fully lightweight components** with a modern rendering system.
Look and Feel	Look and feel depends on the operating system, so it differs across platforms.	Provides **pluggable look and feel**, allowing consistent UI on all platforms.	Provides modern, rich, and highly customizable UI with CSS styling.
Performance	Slower performance due to reliance on native OS components.	Faster than AWT but slower than JavaFX for graphics-rich applications.	High performance with hardware acceleration and advanced graphics support.
UI Design	Limited UI components with basic functionality.	Offers a wide range of advanced UI components like tables, trees, and tabs.	Provides rich UI components, animations, effects, and multimedia support.
Styling Support	Very limited styling options.	Styling is difficult and requires code-based customization.	Uses **CSS**, making UI design easier and more flexible.

Basis	AWT (Abstract Window Toolkit)	Swing	JavaFX
Graphics Support	Basic graphics support.	Improved graphics compared to AWT.	Excellent graphics with 2D/3D support, animations, and visual effects.
Multimedia Support	No built-in multimedia support.	Very limited multimedia capabilities.	Built-in support for audio and video playback.
Ease of Use	Simple but outdated and less flexible.	More powerful than AWT but code can become lengthy.	Easier to design modern UI using FXML, CSS, and Scene Builder.
Architecture	Based on native peer architecture.	Built on top of AWT.	Uses a scene graph architecture for better UI management.
Event Handling	Uses event delegation model but limited features.	Uses improved event handling model.	Uses modern event handling with lambda expressions support.
Platform Dependency	Platform dependent.	Platform independent.	Platform independent.
Current Usage	Rarely used in modern applications.	Still used in legacy applications.	Preferred choice for modern Java GUI development.

(b) i. List any four UI controls in JavaFX. ii. List any two Panes for Containing and Organizing Nodes

(b) i. Four UI Controls in JavaFX

UI controls are components that allow user interaction in JavaFX. Some commonly used controls are:

1. **Button** – Used to perform an action when clicked.
2. **Label** – Displays text or information to the user.
3. **TextField** – Allows the user to enter a single line of text.
4. **CheckBox** – Lets the user select or deselect an option.

Other examples: RadioButton, ComboBox, ListView, Slider, PasswordField

---

(b) ii. Two Panes for Containing and Organizing Nodes

Panes are layout containers used to arrange UI elements (nodes) in JavaFX. Two commonly used panes are:

1. **VBox** – Arranges nodes **vertically** in a single column.
2. **HBox** – Arranges nodes **horizontally** in a single row.

Other examples: BorderPane, GridPane, StackPane, FlowPane

(c) Write a program to demonstrate GUI application development using JavaFX.

```
import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;


public class JavaFXDemo extends Application {


    @Override

    public void start(Stage primaryStage) {

        // Create a label

        Label label = new Label("Welcome to JavaFX GUI!");


        // Create a button

        Button button = new Button("Click Me");


        // Set button action

        button.setOnAction(e -> label.setText("Button Clicked!"));


        // Create a vertical layout (VBox) and add controls

        VBox vbox = new VBox(20); // 20 px spacing
        vbox.getChildren().addAll(label, button);

        // Create a scene with VBox as root node
        Scene scene = new Scene(vbox, 300, 200);

        // Set the stage
        primaryStage.setTitle("JavaFX GUI Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Output

A window opens with a **label** saying *"Welcome to JavaFX GUI!"*

Clicking the **button** changes the label to *"Button Clicked!"*

Q.5 (a) What are the differences between ArrayList and LinkedList? Which list should you use to insert and delete elements at the beginning of a list?

Feature	**ArrayList**	**LinkedList**
Data Structure	ArrayList is implemented using a **dynamic array**, which stores elements in contiguous memory locations.	LinkedList is implemented using a **doubly-linked list**, where each element (node) contains data and references to the previous and next nodes.
Access by Index	Accessing elements by index is **very fast** in ArrayList because it can	Accessing elements by index is **slower** in LinkedList because it needs to traverse nodes

Feature	ArrayList	LinkedList
	directly calculate the memory location of any element.	sequentially from the start or end to reach the desired element.
Insertion/Deletion at Beginning	Inserting or deleting elements at the beginning is **slow** in ArrayList because all subsequent elements need to be **shifted** to maintain order.	Inserting or deleting elements at the beginning is **very fast** in LinkedList because only the **node references** need to be updated.
Insertion/Deletion at End	Adding elements at the end of ArrayList is **fast** in most cases due to dynamic resizing.	Adding elements at the end of LinkedList is also **fast**, since the last node reference is updated.
Memory Usage	ArrayList uses **less memory** because it only stores the actual data without additional references.	LinkedList uses **more memory** because each node stores references to both the previous and next nodes along with the data.
Performance for Search	Searching by index or value is **faster** in ArrayList due to direct access.	Searching by index or value is **slower** in LinkedList because traversal is required.
Best Use Case	ArrayList is suitable when there are **frequent element accesses** and **rare insertions/deletions**.	LinkedList is suitable when there are **frequent insertions and deletions**, especially at the beginning or middle of the list.

Which List to Use for Insert/Delete at Beginning?

- **LinkedList** should be used when you need to **insert or delete elements at the beginning** of a list frequently.
- **Reason:** LinkedList stores elements as nodes connected by pointers, so adding/removing elements at the start **does not require shifting other elements**, making it faster than ArrayList.

(b) What is Java Collection Framework? List four collection classes?

Java Collection Framework (JCF)

The **Java Collection Framework** is a **set of classes and interfaces** that provides **predefined data structures and algorithms** to store, retrieve, and manipulate groups of objects efficiently.

It allows programmers to work with **collections of objects** (like lists, sets, and maps) without writing custom code for basic data structures. The framework provides **standardized interfaces**, concrete classes, and **algorithms** such as sorting, searching, and filtering.

Key Features of Java Collection Framework

1. **Interfaces and Implementations** – Provides a hierarchy of interfaces such as `List`, `Set`, and `Map`, along with their concrete implementations like `ArrayList` and `HashSet`.
2. **Dynamic Data Storage** – Collections can **grow or shrink dynamically**, unlike arrays with fixed size.
3. **Unified API** – All collection classes follow a **common set of methods**, making it easier to use different types of collections.
4. **Algorithms Support** – Built-in algorithms like `sort()`, `reverse()`, and `shuffle()` are available in `Collections` utility class.
5. **Performance** – Optimized for **fast insertion, deletion, and retrieval** of objects.
6. **Type Safety** – Generics ensure that collections can store **specific types** of objects, reducing runtime errors.

Four Common Collection Classes in Java

1. **ArrayList** – Implements the `List` interface; stores elements in a **resizable array**.
2. **LinkedList** – Implements the `List` interface; stores elements in a **doubly-linked list**.
3. **HashSet** – Implements the `Set` interface; stores **unique elements** with no order.
4. **HashMap** – Implements the `Map` interface; stores **key-value pairs** for fast lookup.

Other examples: `TreeSet`, `TreeMap`, `Vector`, `Stack`, `PriorityQueue`

(c) Give the name of class or interface that provides following

facility (Do not elaborate)(one mark for each)

i. Dynamic array to store the duplicate element of different

data types. It maintains the insertion order.

ii. Implements the last-in-first-out data structure.

iii. Implements the first-in-first-out data structure.

iv. Implements the first-in-first-out data structure and it

holds the elements or objects which are to be processed by

their priorities.

v. It represents the unordered set of elements which

doesn't allow us to store the duplicate items.

vi. It is a red-black tree-based implementation. It provides an efficient means of storing key-value pairs in sorted order.

vii. It allows us to store key and value pair, where keys should be unique.

i. **ArrayList**

ii. **Stack**

iii. **Queue** (or **LinkedList** implementing `Queue`)

iv. **PriorityQueue**

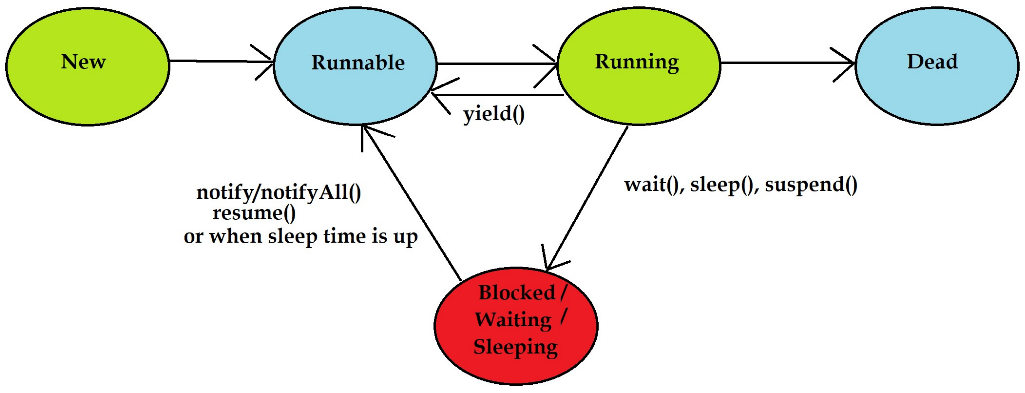
v. **HashSet**

vi. **TreeMap**

vii. **HashMap**

OR

Q.5 (a) Explain thread life cycle.



Thread Lifecycle using Thread states

A **thread** in Java is a lightweight process that executes independently within a program. The lifecycle of a thread is managed by the **Java Virtual Machine (JVM)** and can be represented using **different thread states**.

The main thread states are:

1. New (Created) State

- When a thread object is created using `Thread t = new Thread()`, it enters the **New** state.
- At this stage, the thread is **not yet started** and has not begun execution.
- To start the thread, the `start()` method must be called.

---

2. Runnable State

- After calling `start()`, the thread moves to the **Runnable** state.
- In this state, the thread is **ready to run**, waiting for the CPU scheduler to allocate CPU time.
- The actual running of the thread depends on thread scheduling.
- The thread can also move back to Runnable state from **Blocked/Waiting** state when notified.

---

3. Running State

- When the CPU scheduler selects a thread from the Runnable pool, it enters the **Running** state.
- The thread executes its `run()` method in this state.
- Only **one thread per CPU core** can run at a time.

---

4. Blocked / Waiting / Sleeping State

- A thread moves to this state when it is **temporarily inactive**, waiting for a condition or a resource.
- This can happen due to:
  - `wait()` – waiting for notification from another thread
  - `sleep()` – sleeping for a specified time
  - `suspend()` – manually suspended (deprecated)
- The thread **cannot run** until it is moved back to Runnable state.

---

5. Dead (Terminated) State

- Once the thread **completes execution** of its `run()` method or is stopped, it enters the **Dead** state.
- A thread in this state **cannot be restarted**. A new thread object must be created to run again.

(b) What is the package? Explain steps to create a package with example.

Java Packages

1. What is a Package?

A **package** in Java is a way to **organize classes, interfaces, and sub-packages into a single namespace**. It is similar to a folder in your computer where you keep related files together.

Purpose of packages:

- **Organize classes logically** to avoid clutter in large programs.
- **Avoid name conflicts** between classes with the same name in different packages.
- **Provide access protection** using `public`, `protected`, and default access.
- **Reusability**: Classes in packages can be easily reused in other programs.

Types of packages in Java:

1. **Built-in packages:** Provided by Java, e.g., `java.util`, `java.io`.
2. **User-defined packages:** Created by programmers for specific programs.

---

2. Steps to Create a User-Defined Package

Step 1: Create a class

Write a Java class and save it in a `.java` file.

Step 2: Declare the package

Use the `package` keyword at the **top of the file** before the class definition.

Step 3: Compile the class

Use `javac` with the `-d` option to specify the directory where the package folder should be created.

Step 4: Use the package in another program

Use the `import` keyword to access classes from the package.

---

3. Example of a Package

Step 1 & 2: Create a package and class

```
File: MyPackage/Greetings.java

// Step 2: Declare package
package MyPackage;

// Class inside package
public class Greetings {
    public void sayHello() {
        System.out.println("Hello from MyPackage!");
    }
}
```

Step 3: Compile the class

Open terminal and run:

```
javac -d . MyPackage/Greetings.java
```

- `-d .` tells Java to create the package directory in the current folder.
- This will create a folder `MyPackage` containing `Greetings.class`.

Step 4: Use the package in another program

File: `TestPackage.java`

```
// Import the package
import MyPackage.Greetings;

public class TestPackage {
    public static void main(String[] args) {
        // Create object of package class
        Greetings g = new Greetings();
        g.sayHello();
    }
}
```

Step 5: Compile and run the program

```
javac TestPackage.java
java TestPackage
```

Output:

```
Hello from MyPackage!
```

(c) Declare a class called Employee having employee_Id and

employee_Name as members.

Extend class Employee to have a subclass called

SalariedEmployee having designation and

monthly_salary as members. Define following: - Required constructors - Methods to insert, update and display all details of

employees. - Method main for creating an array for storing these

details given as command line arguments and showing

usage of above methods.

---

1. Employee Class (Superclass)

```java
// Superclass
class Employee {
    int employee_Id;
    String employee_Name;

    // Constructor
    Employee(int id, String name) {
        this.employee_Id = id;
        this.employee_Name = name;
    }

    // Method to insert/update details
    void setDetails(int id, String name) {
        this.employee_Id = id;
        this.employee_Name = name;
    }

    // Method to display details
    void display() {
        System.out.println("Employee ID: " + employee_Id);
        System.out.println("Employee Name: " + employee_Name);
    }
}
```

---

2. SalariedEmployee Class (Subclass)

```java
// Subclass
class SalariedEmployee extends Employee {
    String designation;
    double monthly_salary;

    // Constructor
    SalariedEmployee(int id, String name, String designation, double
salary) {
        super(id, name); // Call superclass constructor
        this.designation = designation;
        this.monthly_salary = salary;
    }

    // Method to insert/update details
    void setSalariedDetails(String designation, double salary) {
        this.designation = designation;
        this.monthly_salary = salary;
    }
    // Override display method
    @Override
    void display() {
        super.display(); // Display superclass details
        System.out.println("Designation: " + designation);
        System.out.println("Monthly Salary: " + monthly_salary);
        System.out.println("----------------------------");
    }
}
```

---

3. Main Class Using Command-Line Arguments

```
public class EmployeeTest {
    public static void main(String[] args) {

        // args example: "101 John Manager 50000 102 Alice Clerk 30000"
        int n = args.length / 4; // Each employee has 4 arguments

        SalariedEmployee[] employees = new SalariedEmployee[n];

        int index = 0;
        for (int i = 0; i < n; i++) {
            int id = Integer.parseInt(args[index++]);
            String name = args[index++];
            String designation = args[index++];
            double salary = Double.parseDouble(args[index++]);

            // Create SalariedEmployee object
            employees[i] = new SalariedEmployee(id, name, designation,
salary);
        }

        // Display all employee details
        System.out.println("Employee Details:");
        for (SalariedEmployee emp : employees) {
            emp.display();
        }

        // Example: update first employee
        if (n > 0) {
            employees[0].setSalariedDetails("Senior Manager", 70000);
            System.out.println("After updating first employee:");
            employees[0].display();
        }
    }
}
```

---

4. How to Run

1. Save the file as `EmployeeTest.java`.
2. Compile:

```
javac EmployeeTest.java
```

3. Run with command-line arguments:

```
java EmployeeTest 101 John Manager 50000 102 Alice Clerk 30000
```

Sample Output:

```
Employee Details:
Employee ID: 101
Employee Name: John
Designation: Manager
Monthly Salary: 50000.0
---------------------------
Employee ID: 102
Employee Name: Alice
Designation: Clerk
Monthly Salary: 30000.0
---------------------------
After updating first employee:
Employee ID: 101
Employee Name: John
Designation: Senior Manager
Monthly Salary: 70000.0
---------------------------
```